# Packet Analysis

This section will focus on peaking into the packets to extract the information (which is what we wanted to begin with). First off we must arm ourselves! Go ahead and get all the relevent RFC's. Lets start off with RFC 791 (IP) RFC 768 (UDP) RFC 826 (ARP) RFC 792 (ICMPv4) and of course RFC 793 (TCPv4) The truth is, once you have these files you dont really need me *sigh* but then again... why right your own code when you can just copy mine! hehe

I would highly recommend you use another packet sniffer to double check your programs... tcpdump will do just fine, and ethereal just kicks ass, you can get either (and more!!) at http://www.tcpdump.org/related.html. Both of these programs are capable of analyzing all fields of a packet, plus the data. Sure we could use them instead of creating our own... but what fun would that be?

I would prefer not to have to rewrite the main body of the program for each new example like I have done previously. Instead I am going to use the same main program and only post the callback function which gets passed to the pcap_loop() or pcap_dispatch() function. Below is a copy of the main program I intend on using (nothing special), go ahead and cut and paste it or download it here.

```
/**********************************************************************
* file:   pcap_main.c
* date:   Tue Jun 19 20:07:49 PDT 2001
* Author: Martin Casado
* Last Modified:2001-Jun-23 12:55:45 PM
*
* Description:
* main program to test different call back functions
* to pcap_loop();
*
* Compile with:
* gcc -Wall -pedantic pcap_main.c -lpcap (-o foo_err_something)
*
* Usage:
* a.out (# of packets) "filter string"
*
**********************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <net/ethernet.h>
#include <netinet/ether.h>

/*
 * workhorse function, we will be modifying this function
 */
void my_callback(u_char *args,const struct pcap_pkthdr* pkthdr,const u_char* packet)
{
}


int main(int argc,char **argv)
{
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    struct bpf_program fp;      /* hold compiled program     */
    bpf_u_int32 maskp;          /* subnet mask               */
    bpf_u_int32 netp;           /* ip                        */
    u_char* args = NULL;

    /* Options must be passed in as a string because I am lazy */
    if(argc < 2){
        fprintf(stdout,"Usage: %s numpackets \"options\"\n",argv[0]);
        return 0;
    }

    /* grab a device to peak into... */
    dev = pcap_lookupdev(errbuf);
    if(dev == NULL)
    { printf("%s\n",errbuf); exit(1); }

    /* ask pcap for the network address and mask of the device */
    pcap_lookupnet(dev,&netp,&maskp,errbuf);

    /* open device for reading. NOTE: defaulting to
     * promiscuous mode*/
    descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
    if(descr == NULL)
    { printf("pcap_open_live(): %s\n",errbuf); exit(1); }


    if(argc > 2)
    {
        /* Lets try and compile the program.. non-optimized */
        if(pcap_compile(descr,&fp,argv[2],0,netp) == -1)
        { fprintf(stderr,"Error calling pcap_compile\n"); exit(1); }

        /* set the compiled program as the filter */
        if(pcap_setfilter(descr,&fp) == -1)
        { fprintf(stderr,"Error setting filter\n"); exit(1); }
    }

    /* ... and loop */
    pcap_loop(descr,atoi(argv[1]),my_callback,args);

    fprintf(stdout,"\nfinished\n");
    return 0;
}
```

I will be using the above program and merely replacing the callback function **my_callback** for demo programs in this section.

Lets start by looking at the datalink headers. "Didn't we already do this", you ask. Sure... sort of, but we didn't spend much time on it so lets just get this out of the way. Looking at the datalink header isn't all too exciting, but it certainly is something we want to stick in our toolkit so we will gloss over the important stuff and continue on. The most important element of the ether header to us is the ether type. Remember **struct ether_header** from **net/ethernet.h**? just so you don't have to click back, here it is again whith the definition of an ether_addr.

```
/* This is a name for the 48 bit ethernet address available on many
   systems.  */
struct ether_addr
{
  u_int8_t ether_addr_octet[ETH_ALEN];
} __attribute__ ((__packed__));

/* 10Mb/s ethernet header */
struct ether_header
{
  u_int8_t  ether_dhost[ETH_ALEN];      /* destination eth addr */
  u_int8_t  ether_shost[ETH_ALEN];      /* source ether addr    */
  u_int16_t ether_type;                 /* packet type ID field */
} __attribute__ ((__packed__));
```

Fortunaty (at least in Linux) **netinet/ether.h** provides us with some fuzzy routines to convert ethernet headers to readable ascii and back..

```
/* Convert 48 bit Ethernet ADDRess to ASCII.  */
extern char *ether_ntoa (__const struct ether_addr *__addr) __THROW;
extern char *ether_ntoa_r (__const struct ether_addr *__addr, char *__buf)
     __THROW;

/* Convert ASCII string S to 48 bit Ethernet address.  */
extern struct ether_addr *ether_aton (__const char *__asc) __THROW;
extern struct ether_addr *ether_aton_r (__const char *__asc,
                                        struct ether_addr *__addr) __THROW;
```

as well as ethernet address to HOSTNAME resolution (that should ring a bell.. :-)

```
/* Map HOSTNAME to 48 bit Ethernet address.  */
extern int ether_hostton (__const char *__hostname, struct ether_addr *__addr)
     __THROW;
```

Previously I pasted some code shamelessly stolen from Steven's Unix Network PRogramming to print out the ethernet header, from now on we take the easy route. Here is a straightforward callback function to handle ethernet headers, print out the source and destination addresses and handle the type.

```
u_int16_t handle_ethernet
        (u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet);

/* looking at ethernet headers */

void my_callback(u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    u_int16_t type = handle_ethernet(args,pkthdr,packet);

    if(type == ETHERTYPE_IP)
    {/* handle IP packet */
    }else if(type == ETHERTYPE_ARP)
    {/* handle arp packet */
    }
    else if(type == ETHERTYPE_REVARP)
    {/* handle reverse arp packet */
    }/* ignorw */
}

u_int16_t handle_ethernet
        (u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    struct ether_header *eptr;  /* net/ethernet.h */

    /* lets start with the ether header... */
    eptr = (struct ether_header *) packet;
    fprintf(stdout,"ethernet header source: %s"
            ,ether_ntoa((const struct ether_addr *)&eptr->ether_shost));
    fprintf(stdout," destination: %s "
            ,ether_ntoa((const struct ether_addr *)&eptr->ether_dhost));

    /* check to see if we have an ip packet */
    if (ntohs (eptr->ether_type) == ETHERTYPE_IP)
    {
        fprintf(stdout,"(IP)");
    }else  if (ntohs (eptr->ether_type) == ETHERTYPE_ARP)
    {
        fprintf(stdout,"(ARP)");
    }else  if (ntohs (eptr->ether_type) == ETHERTYPE_REVARP)
    {
        fprintf(stdout,"(RARP)");
    }else {
        fprintf(stdout,"(?)");
        exit(1);
    }
    fprintf(stdout,"\n");

    return eptr->ether_type;
}
```

You can download the full code here.

Whew! Ok got that out of the way, currently we have a relatively simple framework to print out an ethernet header (if we want) and then handle the type. Lets start by looking at the IP header.

**IP:**
We'll need to wip out our handy dandy RFC's (791 in this case) and take a look at what it has to say about IP headers... here is a copy of the section which decsribes the header.

```
3.1 Internet Header Format

       A summary of the contents of the internet header follows:

           0                   1                   2                   3
           0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |Version|  IHL  |Type of Service|          Total Length         |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |         Identification        |Flags|      Fragment Offset    |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |  Time to Live |    Protocol   |         Header Checksum       |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |                       Source Address                          |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |                    Destination Address                        |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          |                    Options                    |    Padding    |
          +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                           Example Internet Datagram Header

                                      Figure 4.

         Note that each tick mark represents one bit position.
```

Now lets peak at **netinet/ip.h**

```
struct ip
```

```
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ip_hl:4;               /* header length */
    unsigned int ip_v:4;                /* version */
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int ip_v:4;                /* version */
    unsigned int ip_hl:4;               /* header length */
#endif
    u_int8_t ip_tos;                    /* type of service */
    u_short ip_len;                     /* total length */
    u_short ip_id;                      /* identification */
    u_short ip_off;                     /* fragment offset field */
#define IP_RF 0x8000                    /* reserved fragment flag */
#define IP_DF 0x4000                    /* dont fragment flag */
#define IP_MF 0x2000                    /* more fragments flag */
#define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
    u_int8_t ip_ttl;                    /* time to live */
    u_int8_t ip_p;                      /* protocol */
    u_short ip_sum;                     /* checksum */
    struct in_addr ip_src, ip_dst;      /* source and dest address */
  };
```

Cool, they seem to match up perfectly.... this of course would be fine to use, but I prefer to follow the tcpdump method of handling the version and header length.

```
struct my_ip
        u_int8_t        ip_vhl;         /* header length, version */
#define IP_V(ip)        (((ip)->ip_vhl & 0xf0) >> 4)
#define IP_HL(ip)       ((ip)->ip_vhl & 0x0f)
        u_int8_t        ip_tos;         /* type of service */
        u_int16_t       ip_len;         /* total length */
        u_int16_t       ip_id;          /* identification */
        u_int16_t       ip_off;         /* fragment offset field */
#define IP_DF 0x4000                    /* dont fragment flag */
#define IP_MF 0x2000                    /* more fragments flag */
#define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
        u_int8_t        ip_ttl;         /* time to live */
        u_int8_t        ip_p;           /* protocol */
        u_int16_t       ip_sum;         /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
```

Lets take a first stab at peaking into the IP header... consider the following function (full source here).

```
u_char* handle_IP
        (u_char *args,const struct pcap_pkthdr* pkthdr,const u_char*
        packet)
{
    const struct my_ip* ip;
    u_int length = pkthdr-&len;
    u_int hlen,off,version;
    int i;

    int len;

    /* jump pass the ethernet header */
    ip = (struct my_ip*)(packet + sizeof(struct ether_header));
    length -= sizeof(struct ether_header);

    /* check to see we have a packet of valid length */
    if (length < sizeof(struct my_ip))
    {
        printf("truncated ip %d",length);
        return NULL;
    }

    len     = ntohs(ip->ip_len);
    hlen    = IP_HL(ip); /* header length */
    version = IP_V(ip);/* ip version */

    /* check version */
    if(version != 4)
    {
      fprintf(stdout,"Unknown version %d\n",version);
      return NULL;
    }

    /* check header length */
    if(hlen < 5 )
    {
        fprintf(stdout,"bad-hlen %d \n",hlen);
    }

    /* see if we have as much packet as we should */
    if(length < len)
        printf("\ntruncated IP - %d bytes missing\n",len - length);

    /* Check to see if we have the first fragment */
    off = ntohs(ip->ip_off);
    if((off &apm; 0x1fff) == 0 )/* aka no 1's in first 13 bits */
    {/* print SOURCE DESTINATION hlen version len offset */
        fprintf(stdout,"IP: ");
        fprintf(stdout,"%s ",
                inet_ntoa(ip->ip_src));
        fprintf(stdout,"%s %d %d %d %d\n",
                inet_ntoa(ip->ip_dst),
                hlen,version,len,off);
    }

    return NULL;
}
```

Given a clean arp cache this is what the output looks like on my machine, when I try to telnet to 134.114.90.1...

```
[root@localhost libpcap]# ./a.out 5
ETH: 0:10:a4:8b:d3:b4 ff:ff:ff:ff:ff:ff (ARP) 42
ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (ARP) 60
ETH: 0:10:a4:8b:d3:b4 0:20:78:d1:e8:1 (IP) 74
IP: 192.168.1.100 134.114.90.1 5 4 60 16384
ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (IP) 60
IP: 134.114.90.1 192.168.1.100 5 4 40 0
```

Lets try and reconstruct the conversation shall we?

- **my computer:** Who has the gateways IP (192.168.1.100)?
  ETH: 0:10:a4:8b:d3:b4 ff:ff:ff:ff:ff:ff (ARP) 42
- **gateway:** I do!!
  ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (ARP) 60
- **my computer(through gateway):** Hello Mr. 134.114.90.1 can we talk?
  ETH: 0:10:a4:8b:d3:b4 0:20:78:d1:e8:1 (IP) 74 IP: 192.168.1.100 134.114.90.1 5 4 60 16384
- **134.114.90.1:** Nope, I'm not listening
  ETH: 0:20:78:d1:e8:1 0:10:a4:8b:d3:b4 (IP) 60 IP: 134.114.90.1 192.168.1.100 5 4 40 0

I have admittedly skipped TONS of information in a rush to provide you with code to display the IP header (thats all you really wanted anyways wasn't it :-). That said, if you are lost don't worry, I will slow down and attempt to describe what exactly is going on. All that you really need to know up to this point is..

- All packets are sent via ethernet
- The ethernet header defines the protocol type of the packet it is carrying
- IP is one of these types (as well as ARP and RARP)
- The IP header is confusing ...

So before getting too far into packet dissection it would probably benefit us to regress a bit and talk about IP...
"awww but.... that sounds boring!",you say. Well if you are really anxious I would suggest you grab the tcpdump source and take a look at the following methods ... :-)

- ether_if_print (print-ether.c)
- ip_print (print-ip.c)
- tcp_print (print-tcp.c)
- udp_print (print-udp.c)

I've also found the sniffit source to be a great read.

[prev] [next]